Run-time Phenomena in Dynamic Software Updating: Causes and Effects

Gregersen, Allan Raundahl; Jørgensen, Bo Nørregaard

Published in:
Proceedings of the joint meeting of the 12th International Workshop on Principles on Software Evolution and the 7th ERCIM Workshop on Software Evolution IWPSE-EVOL 2011

DOI:
10.1145/2024445.2024448

Publication date:
2011

Document Version
Final published version

Run-time Phenomena in Dynamic Software Updating: Causes and Effects

Allan Raundahl Gregersen
Maersk McKinney Moller Institute
University of Southern Denmark
Campusvej 55, DK-5230 Odense M
+45 65 50 35 77
allang@mmmi.sdu.dk

Bo Nørregaard Jørgensen
Maersk McKinney Moller Institute
University of Southern Denmark
Campusvej 55, DK-5230 Odense M
+45 65 50 35 45
bnj@mmmi.sdu.dk

ABSTRACT
The development of a dynamic software updating system for statically-typed object-oriented programming languages has turned out to be a challenging task. Despite the fact that the present state of the art in dynamic updating systems, like JRebel, Dynamic Code Evolution VM, JVolve and Javeleon, all provide very transparent and flexible technical solutions to dynamic updating, case studies have shown that designing dynamically updatable applications still remains a challenging task. This challenge has its roots in a number of run-time phenomena that are inherent to dynamic updating of applications written in statically-typed object-oriented programming languages. In this paper, we present our experience from developing dynamically updatable applications using a state-of-the-art dynamic updating system for Java. We believe that the findings presented in this paper provide an important step towards a better understanding of the implications of dynamic updating on the application design.

Categories and Subject Descriptors
D.2.3 [Software Engineering]: Coding Tools and Techniques, D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; C.4 [Performance of Systems]: Reliability, Availability, and Serviceability.

General Terms
Design, Reliability, Experimentation.

Keywords
Dynamic software updating, on-line software evolution, dynamic updating, run-time evolution.

1. INTRODUCTION
The ability to dynamically update the code of a running application is interesting to many domains, as it can effectively reduce system downtime by eliminating the need for halting the application, reinstalling the new version, and restarting the application again. However, as promising as it may sound, dynamic software updating has been underway for the past decades, without making its breakthrough in mainstream application development. The main reason seems to be the absence of an approach that matches the workflow of the typical application developer. Recently, dynamic updating approaches based on dynamic reloading of Java classes have gained popularity, especially as tools for boosting productivity during the development phase. State-of-the-art approaches for Java includes; JRebel, [17] an application-level system which is currently the only commercial tool for class reloading in Java; Dynamic Code Evolution VM, [32] a VM-enhancement of the Java HotSwap [8] functionality; JVolve, [29] a solution based on the Jikes Research VM, and Javeleon, [11] a dynamic updating system defined at the application-level. In this paper, we outline commonalities and differences of those systems, hereby setting the stage for our investigations of how language-transparent dynamic updating systems affect application design and vice versa. We say that a dynamic updating system is language transparent, if it does not extend the target language and that programmers are allowed to make use of the entire language. This paper shifts the focus towards the next major challenge in the field of dynamic software updating, namely how to design applications that are dynamically updatable using unconstrained dynamic class redefinition. Hence, the main goal of the work presented here is to investigate to what extent it is possible for developers to retain their current design and coding practices in the development of dynamically updatable applications. For this reason, we have conducted a series of controlled case studies, to gain a better understanding of the implications of using a dynamic updating system designed for a modern statically-typed class-based object-oriented programming language. These case studies have revealed a number of run-time phenomena whose root causes stem from the concrete application designs and coding practices. As we shall see in this paper, the particular use of language features and design styles has a significant impact on the dynamic updatability of running applications. Hence, one of the main contributions of this paper is to show general examples of how a language-transparent dynamic updating system, is not inherently transparent to the application design. In addition, the revealed phenomena have pushed the borders for what we would normally consider correct application behavior. This has led us to question whether we should expect the same run-time behavior of a dynamically updated application as we can from an application that undergoes the traditional halt, redeploy and restart cycle. Therefore, in this paper we investigate the following questions:

- *To what extent does an application's design affect its ability to be dynamically updated?*
- *How does dynamic updating influence our perception of correct behavior of an updated application?*

To address these questions, we discuss previous efforts in dynamic updating in section 2. In section 3, we present our dynamic updating system Javeleon while comparing it to other state-of-the-art dynamic updating systems. Section 4 introduces the observed run-time phenomena. In section 5, we identify atomic code changes and common code refactoring for which these phenomena may surface.

IWPSE-EVOL'11, September 4, 2011, Szeged, Hungary.

In section 6, we discuss the implications of dynamic updating on application design. We conclude upon our findings in section 7.

2. STATE OF THE ART

Several approaches to dynamic updating have been proposed for both procedural, e.g., [1],[7],[9],[15],[19], and class-based object-oriented languages, e.g., [2],[8],[16],[17],**Error! Reference source not found.**[22],[23],[25],[29],[32]. Common to these approaches is a clear trend to focus solely on researching the technical issues of constructing a general-purpose dynamic updating system, and not on its actual application. Logically, this trend has given rise to a significant number of studies focusing on particular technical capabilities of prototype systems. However, the focus on technical issues has resulted in the absence of work studying the implications of using a dynamic updating system in praxis. This is especially true for statically-typed object-oriented languages, where there is no comprehensive understanding of the run-time effects that the presence of a dynamic updating system imposes for a specific application design. Thus, there is no evidence that precludes the possibility that certain application design principles may perform better than others during dynamic updating. In the following, we will discuss the applicability of existing dynamic updating systems for investigating the stated research questions.

State-of-the-art systems for procedural languages include GINSENG, [19], DSU, [15] and POLUS, [7]. GINSENG proposes to apply flexible changes to a running system by means of patches derived semi-automatically. In the same vein, KSplice, [1] an approach for dynamically updating functions in procedural languages, has been successfully applied to incorporate security patches for the Linux kernel at runtime. Much effort has been spent on ensuring the safety and robustness of these approaches; e.g. [20], [28], [14]. In particular, the work in [14] deals with the issue of robustness, by presenting a novel test-based approach to evaluate dynamic update correctness. By performing dynamic updates continuously during a test phase, valuable information can be extracted pertaining to applying different policies for inducing safe update points. Their experiments show that performing dynamic updates at arbitrary update points without any safety checks produces many faults. While this work seems particularly useful for developers to validate the updateability of a new version with respect to the previous one, it remains unclear how it would scale when updateability must be ensured for multiple revisions of the underlying application. It may very well show that a particular "safe" update point in a base version, while ensuring correct behavior for a specific update (because it passes the tests), may fail for subsequent updates. While further transactional consistency properties, as suggested by [21], ensure that user-defined transactional code appears to execute entirely in the same code version, there are still a number of, so far, uninvestigated run-time phenomena that can surface after performing dynamic updates regardless of the presence of safe update points and controlled transactions. However, safe update points and transaction support do contribute to robustness, but as we will describe later, they are not sufficient in all cases. This paper focuses on the causes and effects of exactly these cases.

Previous efforts for object-oriented languages have mainly been focused on the ability to update applications written in a statically-typed language. The main reason for this tendency is that the type system is the single, most influential aspect in dynamic updating, [30]. In dynamically-typed languages, there is typically reflective support for assignability between multiple versions of objects. On the contrary, a static type system will enforce static type safety by disallowing any type conversions between multiple versions of a class, thereby setting up a version barrier, [26].

Many approaches targeting statically-typed object-oriented languages have shown great potential, e.g., [8],[16],[17],[22],[23], [29]. However, some of these systems compromise language transparency by imposing safe update points, [29], or by forcing developers to comply with a certain programming style or architecture, [16]. However, some existing dynamic updating systems are largely language transparent, [8],[17],[22],[23], including our own Javeleon. Another important property is flexibility, i.e., the code changes that the dynamic updating system supports. Compromising flexibility will effectively reduce the set of observed run-time phenomena. Beginning at the lower end of the flexibility scale, we find the currently built-in HotSwap mechanism, [8]. HotSwap permits changes to already declared method bodies only. More flexibility is added by approaches that allow changes to class members that do not break the public class interface, [16],[22]. Other approaches even support changes to the public class interface, [17],[23],[29]. Among them is JVolve, which introduces dynamic updating at the level of the VM. JVolve has zero steady-state overhead making it an attractive alternative for developers. However, the lack of support for dynamically modifying the class inheritance hierarchy renders these approaches too restrictive for investigating the full implications of using dynamic updating. Undeniably, a problematic issue when using these approaches is that they force developers to introduce workarounds in order to circumvent the shortcomings. Hence, these approaches may unintentionally encourage developers to make poor design decisions and/or foster code smells. Nonetheless, JRebel, [17] has shown that a class reloading mechanism, in which inheritance changes are not permitted, is still very useful especially within development scenarios. Recently, another promising candidate, the Dynamic Code Evolution VM (DCEVM), [32] has entered the scene. DCEVM is a language-transparent dynamic updating system permitting changes to the type hierarchy on par with Javeleon in this matter. In our case study experiments, we have chosen to use our own system Javeleon mainly because of 1) JRebel and JVolve do not provide the support we need to understand the effect of changing the inheritance hierarchy, and 2) DCEVM was not publically available at the time the experiments were made. Moreover, DCEVM has a few shortcomings as will be described in the detailed comparison in the next section. However, we would like to point out that using DCEVM to conduct similar experiments would result in the same phenomena as we have found using Javeleon. That is, the run-time phenomena treated in this paper are general for all approaches.

3. COMPARISON OF DYNAMIC UPDATING SYSTEMS

This section compares the main features of the said candidate dynamic updating systems. Technical details about the inner workings of Javeleon will not be discussed as these have already been comprehensively explained in our previous work, [11], [12].

One of the main objectives in realizing a transparent dynamic updating system is to let developers retain their usage of the Java language and the way they design applications. This purity provides a way to investigate whether different ways of using the language features or designing one's application would affect the outcome of the dynamic updating process. Table 1 captures the main features of JRebel, Javeleon, DVEVM and JVolve.

Table 1. Dynamic updating features.

	JRebel	Javeleon	DCEVM	JVolve
Changes to method bodies	✓	✓	✓	✓
Adding/removing fields	✓	✓	✓	✓
Move field to super/sub class (state preserving)	✗	✓	⚠	?
Changing static field value	⚠	✗	✗	✗
Changing static final field value	✓	✓	✓	✓
Automatic initialization of new fields	✗	⚠	✗	✗
Adding/removing methods	✓	✓	✓	✓
Adding/removing constructors	✓	✓	✓	✓
Adding/removing classes	✓	✓	⚠	⚠
Changing interfaces	✓	✓	✓	✓
Adding/removing enum values	✓	✓	✗	?
Replace superclass	✗	✓	⚠	✗
Adding/removing implemented interfaces	✗	✓	✓	✗
Reloading of resources	⚠	⚠	✗	✗

All approaches listed in Table 1 allow dynamic type-safe updates of code for method bodies and member fields. They are also capable of persisting application state. JVolve and DCEVM handle the state migration at update time by pausing the entire VM, whereas Javeleon provides a thread-safe lazy state migration mechanism thus allowing for non-blocking updates as does JRebel. JRebel does not support state-preserving move field refactoring in the class hierarchy, which is supported by Javeleon and DCEVM. However, DCEVM currently incorrectly copies field values to super/sub classes even if the field is retained in the original class. JRebel is the only approach with support for changing static field values. However, it is based on re-executing the entire static initializer which could lead to serious side effects caused by multiple executions of code where only one is expected. Javeleon supports automatic field initialization without re-executing the entire constructor/static initializer. A minor limitation to Javeleon's automatic initialization is that tertiary expressions may not be used as part of the field assignment. For VM approaches the task of adding a new class into a running system is not easy because it requires addition of a protocol to specify in which particular class loader the new class must be defined. This is not an issue for standalone Java applications, but adding a component system with a custom-class-loader hierarchy into the mix makes it nearly impossible to perform this task without specific integration with the component system. Currently, the DCEVM and JVolve are not able to support such class additions. Only Javeleon and DCEVM are able to replace the super class of a class. However, the mechanism used in DCEVM may result in VM crashes after resuming execution. Finally, reloading of resources requires specific integration with frameworks, application servers and component systems, which JRebel does well for a wide range of different frameworks. Javeleon currently only integrates with the NetBeans Platform.

All approaches use a simple one-to-one state migration policy that transfers the value of a field in the former class version to the same field (possibly moved to super/sub class) in the new class version. Consequently, if an update changes the representation of an object's state in an incompatible manner, i.e., by changing the type of a field, the field value (state) is lost. Using explicit state transfer functions (STF) as suggested in, e.g., [19] is known to solve this problem, as it can perform the necessary conversion between types. Javeleon, JRebel and DCEVM have built-in mechanisms to support explicit state transfer, by means of special methods that are invoked by the updating system on every updated class and all instances of the class before they are provided for usage in the updated program. However, we argue that using these special methods should be a last resort, since the complexity will go up. We believe that by understanding the phenomena in this paper and by using a dynamic updating system without focusing on STF it will be possible to provide an adequate learning tool to avoid extensive usage of STF. In addition, we argue that STF should not be used as a patch coming from one version to another, but rather as the means to let objects reflect upon themselves and adapt accordingly regardless of their origin. This is an important distinction from traditional patching.

4. RUN-TIME PHENOMENA IN DYNAMIC UPDATING

This section discusses run-time phenomena that may surface as an effect of performing dynamic updates. Here, phenomena are any run-time effect which would not have been observed if the new version of the application was started from scratch. Explicit attention is drawn to the root causes for why these phenomena occur, and how they may be avoided, by suggesting simple changes to the design and/or the implementation. The discussed run-time phenomena reoccur in all of our case studies on dynamic updating. In this section, we will use a running example derived from a recent case study in game development. Game development provides a good platform for the experiments, because large portions of a game's internal state have some form of visual counterpart in the game's GUI. Due to this relationship between state and GUI, the discussed phenomena will have some form of visual manifestation. The case study was designed to include a few major revisions of increasing functional complexity. This was done to ensure a realistic test setup. All revisions are placed under version control, to keep track of code changes that may cause a particular phenomenon. For interested readers, a subversion repository dump file together with instruction for use is available at http://javeleon.org/?demo. Code inspection was used to identify the exact code changes that cause a given phenomenon. The next subsections discuss the various forms of the run-time phenomena that were observed when dynamically updating the game through its consecutive versions.

4.1 Phantom Objects

Phantom Objects are live objects whose classes have been removed by a dynamic update. While phantom objects will continue to exist in the system, their existence in the updated application will be invalidated. Hence, if such objects are part of the existing application state, the updated application may try to reference them indirectly through, e.g., a collection. Although removing classes is typically discouraged, there are situations where classes are either renamed (for an automatic updating system this corresponds to a

class removed- and a class added operation) or in-lined due to refactoring. Likewise, the use of dayfly classes [18] is a good example of class removals. Dayfly classes are classes that are typically created for evaluating a new idea and then removed shortly thereafter. We first spotted this phenomenon when using Javeleon to downgrade/roll back the Breakout Game. It turned out that a new feature to support special feature bricks, such as concrete bricks or bonus bricks that drop bonuses when hit, was modeled by several classes all being subclasses of an abstract parent class (Brick.class). The resulting effect is depicted in Figure 1, showing how all special bricks (all but the purple bricks) disappeared after the roll-back update.

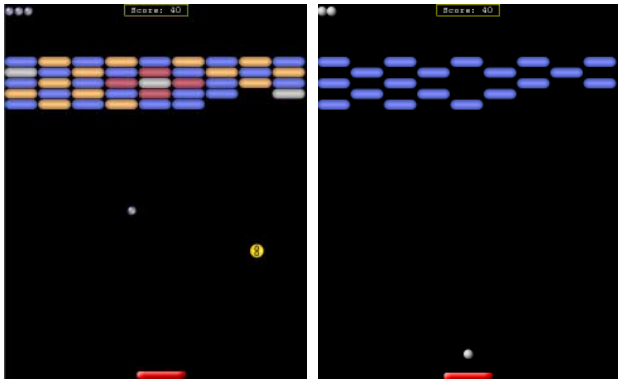

Figure 1. Disappearing objects after class removals.

The run-time effect of removing a class depends highly on the design of client classes. For example, if the bricks in the brick wall were maintained in an array (Brick[][]), and some client code were to check for collisions with a ball by iterating the bricks in the brick wall, this code would try to touch phantom objects (instances of removed subclasses of Brick), which would result in a application terminating null pointer exception (NPE). NPE will occur because the lazy state migration mechanism in Javeleon will insert a null reference in place of the original special brick object, because it will not be able to map this phantom object to a type in the new version. Hence, designs iterating over arrays and collections are fragile to dynamic class removals. A more robust design would be to apply a loosely-coupled design, like the Lookup in the NetBeans Platform, [4]. The client code would then be written in terms of looking up bricks that can be understood by the current caller as exemplified in Code Snippet 1.

```
for (Brick brick : localLookup.
lookupAll(Brick.class)) {
    checkCollision(someBall,brick);
}
```

Code Snippet 1. Loosely-coupled design.

If this code snippet was invoked before the dynamic update, the `lookupAll()` invocation would locate all the bricks as seen in the leftmost screenshot in Figure 1. Now, executing the code after the dynamic update would bring us to the scenario depicted by the rightmost screenshot in Figure 1, without causing an NPE. However, it is still questionable to what extent the scenario in Figure 1 corresponds to correct application behavior. Not only is the representation of the bricks in the rightmost screenshot impossible to reach if the application had been restarted with the new code, but the relation between the number of bricks in the wall and the score will be almost impossible to maintain in this situation. So the resulting question is: can we provide tools to handle phantom objects? One solution would be to include a dynamic verifier detecting (at update time) the presence of live instances of removed classes and simply abandon the update if any such instances were found. Indeed, this would increase the robustness of the dynamic updating system, but nonetheless also compromise the flexibility of the system. Another solution would be to use state transfer functions (STF) to bring the system into a consistent state. However, as the explicit use of STF compromise transparency, we advocate application designs that tolerate less strict correctness criteria before resorting to STF.

Another variant of the phenomenon phantom objects is *Phantom State*, meaning a field that holds a value whose dynamic type is removed by a dynamic update. Obviously, trying to use this field directly after the update will throw an NPE. A logical design choice that would eliminate this phenomenon would be to enforce self-encapsulation, [10] and lazy initialization, [3], as exemplified in Code Snippet 2. Not only would this simple design idiom provide a resilient mechanism for getting rid of Phantom State, but it also offers an implicit mechanism for optional state transfer functions. In addition, self-encapsulating fields are known to ease maintenance as well as provide for a more extensible design, especially when working with inheritance hierarchies, [10].

```
ReturnType getTheValue() {
    if(theField == null)) {
        theField = new ReturnType(...);
    }
    Return theField;
}
```

Code Snippet 2. Self-encapsulation and lazy field initialization.

Being aware that developers would not always remember to enforce this design pattern, or perhaps some do not like this particular coding style, Javeleon offers the special state transfer function that can be used to perform consistency checks in a well-defined place.

4.2 Transient Inconsistency

During a *Transient Inconsistency*, an updated application is temporally in an unreachable run-time state, i.e., a state which the new version of the application would never enter if it were started from scratch. If an application does not arrive at a reachable run-time state after a finite period of execution, it is said to be captured in an erroneous state. Another variant of the Transient Inconsistency phenomenon is the *Transient State Inconsistency*, where the relationships between a set of fields are different in the new version of the application. Hence, a set of otherwise consistent state fields will be invalidated by the dynamic update as the previous assumptions about their mutual interrelationships are changed. A good example of a transient state inconsistency can be derived from the Breakout Game's score system. Let's say that the score is calculated based on the number of bricks hit. Consider what would happen if we were to double the number of points per broken brick in a level. This would put us in a transient inconsistent state because the ratio between the number of currently broken bricks and the score would not have been possible if the updated revision was started from scratch. Obviously, a dynamic updating system cannot correct such state inconsistency automatically. Traditionally, the suggestion would be to incorporate an STF to fix such issues. Instead of doing so right away, we argue that we should rethink the way we design. For this particular case a simple change to determine the score based on the remaining bricks would suffice, as this brings the transient state inconsistency to an end when the next score calculation is made. Hence, a change to the perception of the current application state requires a design that is capable of correctly deducing the new application state by inspecting the currently live objects in the application, and not by relying on static assumptions which may no longer hold. Unfortunately, a change to

the application design will not always suffice to overcome transient inconsistencies. Consider the run time addition of an "X2 bonus" activated at some point in the game. It is simply impossible for a transparent dynamic updating system to infer when or if this bonus was activated and how many bricks were hit during the "X2 bonus" period. The most important lesson to learn here is that semantic changes that depend on events which may or should have happened, at an already passed time, according to the new program specification, are very difficult, if not impossible, to handle automatically. In such situations, the developer has to either 1) accept the transient inconsistency, which may be reasonable if the transient inconsistency does not compromise correctness of the overall application behavior, 2) write an appropriate STF for determining if the transient inconsistency will manifest itself in the current setting and, if so, abandoning the current game/session/transaction or 3) perform a full application restart or, if possible, reload the sub-part of the application affected by the change (acceptable in development mode if writing the STF is time-consuming).

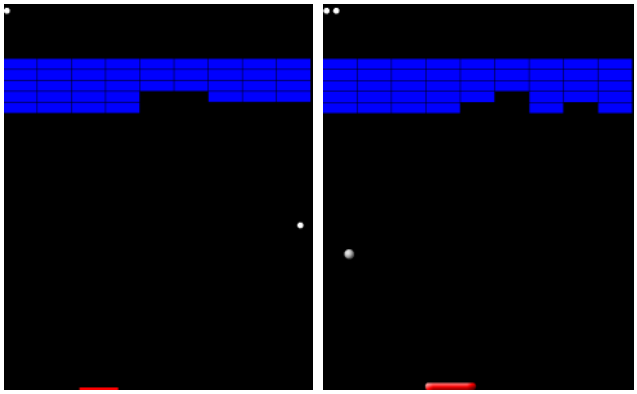

Figure 2. Delayed update of the brick wall.

Another interesting variant of transient inconsistency is *Delayed Effect*. When a delayed effect occurs it means that the expected effect of a dynamic update does not immediately appear, because it depends on some other event in the system, such as when a screen is redrawn. During a delayed effect, the application will normally be in an unreachable state. A delayed effect is typically caused by programming techniques such as screen buffering and data caching. Figure 2 shows an example of a delayed effect in the Breakout Game. The leftmost picture in Figure 2 captures the graphical representation of the initial game version where GUI elements are painted by simple solid color fills. In the rightmost picture, which is captured just after applying a dynamic update, we see that the visual appearance of the paddle and the ball has been changed, but the bricks in the brick wall have not yet been updated to look like the bricks previously depicted in Figure 1. The reason is that the chosen paint strategy implements a lazy repaint mechanism that only repaints elements when they are moved or hit by other elements.

4.3 Oblivious Update

The *Oblivious Update* phenomenon is the absence of an expected run-time effect that would have occurred if the system was started from scratch. For example, changing a constructor to initialize new state fields will not have any effect on already created instances, as dynamic updating systems should not re-run constructors. Re-running constructors will not only overwrite the current state of existing objects, but it may also corrupt the whole application, if the constructor contains code with side-effects.

To illustrate the nature of an oblivious update let us look at how the Hi-Score facility is added to the Breakout Game. In the earlier versions of the game a central 'Game' class controlled the current score object which was just discarded/reset on game over events. Since we aimed for reusability, we decided to put in an event-based lifecycle controller that facilitated registration and notifications to lifecycle event observers. In the revised game version. a Hi-Score object is registered as a listener to, e.g., game over events as shown in Code Snippet 3.

```
// version 1
Class Game {
    Private Score currentScore;
    Game() { /* initialize score */ }
}
// version 2
class Game {
    Private Score currentScore;
    Game() { /* initialize score */
        Events.addGameListener(
            new HiScore());
    }
}
    // code executed on game over event
    Events.notifyGameOver(currentScore);
```

Code Snippet 3. Adding the Hi-Score to the Breakout Game

While this new feature would be easily picked up by restarting the application, this is not the case using Javeleon. The problem is that the registration code is placed inside a constructor that has already been executed. If the update shown in Code Snippet 3 is carried out, no listener object will be associated with the existing game object, because the constructor will not be re-executed. The net result is that a Hi-Score cannot be achieved in the current game. We say that the expected effect of the update is absent. This particular example could also qualify as a transient inconsistency depending on the behavior of the Game object. If a new Game object is constructed before a new game, then we would face a transient inconsistency, since the Hi-Score would be absent only during the current game. On the other hand, an oblivious update will manifest if the Game object is reset and reused throughout the applications lifetime. The preferable solution, is to utilize a better design based on loose coupling. The Game class should not really know about the HiScore, but merely be responsible for the lifecycle of the game notifying events when appropriate. This kind of loosely coupled communication can easily be realized with the Lookup mechanism. Hence developers need only to define the new HiScore class implementing the appropriate interface used by the general event handler in this application. Then the only change needed in the Game class is to notify the event handler when lifecycle events occur. This loosely-coupled design not only improves dynamic updatability but also increases the extensibility, since it would be easy to add new features taking action on game lifecycle events without having to change the Game class at all. However, if an application is poorly designed as is the case with the Breakout Game, the use of STF is unavoidable in some cases. The use of STF requires careful consideration as they can cause severe flaws in the production code, since developers may forget to implement the actual changes required for the code to function properly after a complete restart of the system. So trying to spot an extensible application design that will not require the use of STF is highly recommended.

Another variant of oblivious update of a more technical nature is observed when applying dynamic updating to applications that are based on component or web frameworks of some kind. These frameworks typically provide the means to inject functionality into the system declaratively. Take for instance the NetBeans Platform,

[4] (the core platform which the NetBeans IDE is based upon) as an example. NetBeans permits developers to build the entire menu declaratively. Obviously, a dynamic updating system must be able to refresh such declarative injections during a dynamic update, if they have been changed. The current reference implementation of Javeleon has full NetBeans Platform integration, which means that it rescans the declarative information and publishes the result immediately after a dynamic update, thus avoiding any oblivious updates of this nature.

4.4 Broken Assumption

Whenever the soundness of the value of a member field (e.g. a counter) depends on some other member field (e.g. constant value), changing the logic of the code that maintains this interdependency may break objects of the class. Consider the example in Code Snippet 4.

```
private int observations = 0;
private static final int MAX_OBSERVATIONS = 10;
void checkObservations() {
    if(observations++ == MAX_OBSERVATIONS)
        alert();
}
```

Code Snippet 4. The soundness of one field depends on another. Here the observations field is valid within [0;10].

Now consider what happens if we lower the constant value MAX_OBSERVATIONS to 7 in the new revision. In case the value of the observations field was seven or more at update time, the new version will never fire an alert. The observant reader may wonder why Code Snippet 4 does not enforce self-encapsulation of the observations field as advocated in section 4.1. In fact, that would provide a solution.

```
private int observations = 0;
private static final int MAX_OBSERVATIONS = 10;
private int getObservations() {
    if(observations >= MAX_OBSERVATIONS)
        observations = MAX_OBSERVATIONS;
    return observations;
}
void checkObservations() {
    setObservations(getObservations()++);
    if(getObservations() == MAX_OBSERVATIONS)
        alert();
}
```

Code Snippet 5. Using self-encapsulation and lazy initialization to overcome broken assumptions.

Now, if we re-wrote the code to that of Code Snippet 5 we would end up with the expected semantics. However, this will significantly bloat the code with checks that are only relevant to a dynamic updating system. Therefore we might as well use STF to handle the inconsistency.

4.5 Lost State

A *Lost State* phenomenon surfaces in case an updated class makes binary incompatible changes to the type of a member field. Given that Javeleon transfers state automatically, it is not possible to deduct how a changed type relates to a previously declared type. The run-time effect of changing the field type is that the field value for all existing objects of that class is lost and the new value is set to the default value. However, developers using Javeleon have options to either recover the old field value or to re-initialize the field to a new value appropriate for the new class version, by following simple design principles. Recovering the old field value can be used to construct the new field value, e.g., when changing a field type from a String to a user-defined class that encapsulates the string value, as used in the 'Replace Data Value With Object' refactoring, [10]. As Javeleon has been developed with language transparency in mind, there is no language construct in place to deal with field type changes, or for that matter field renaming; however, such support as seen in, e.g., Revision Classes, [24] could be added to aid the developers in this task. With no language support the developers have to define a completely new field and leave the old field intact. Then, utilizing the automatic initialization of new fields in Javeleon, the developer can write the type conversion code as shown by Code Snippet 6.

```
Private String email; // version 1
// version 2
Private String email;
Private Email emailAddr = convertToEmail(email);
```

Code Snippet 6. Changing the type of a field.

Another variant of the phenomenon is *Absent State,* defined as the situation in which objects or classes having been created in a previous version, once migrated to the new version, lacks a portion of the expected state. While the Lost State phenomenon refers to a specific application state in the former version, which is lost after the update, the Absent State phenomenon refers to a state that would have been created, e.g., by using an extra argument in a modified constructor. Examples of the Absent State phenomenon include the addition of a person's SSN in an updated Person class. Although the Absent State phenomenon is similar to the Oblivious Update phenomenon, there is a clear distinction. An oblivious update will allow the updated application to continue execution without causing run-time exceptions, while an absent state will cause null pointer exceptions when the updated code tries to reference the missing data. In an application using a database system, such updates to a database schema are always assumed to be properly handled by the updated application logic. Here, developers immediately accept that part of an application's state is absent. Once developers realize this fact, they will quickly adapt to writing robust application logic that takes absent state into account even for dynamically evolvable applications. The suggested best practice to use self-encapsulation and lazy initialization provides the necessary tool for developers to include means to correct the absent state. Section 6 will include further discussion on the similarities with database schema evolution.

5. RUN-TIME EFFECTS OF ATOMIC CODE CHANGES AND REFACTORINGS

Having explained how a number of run-time phenomena can surface, and how the underlying design in many cases can be improved to remedy these phenomena, this section turns to a more precise view on what causes these phenomena in relation to atomic code changes and common code refactoring.

Our study on atomic code changes is based on the classification of code changes given in [15]. From the set of all 104 atomic code changes, we have identified 11 (merged to 9 in Table 2) code changes that can cause the discussed run-time phenomena.

Class removed. Corresponds directly to that of Phantom Object.

Class added. Adding a new subclass of a class with the purpose of differentiating between instances of the class may result in *Absent State*, as all live instances are automatically migrated to the new version of the class, including those that should have been promoted to instances of the new subclass. Hence, the expected instances of the newly-added subclass are absent.

Modifier abstract added to class. Adding abstract to a class invalidates live instances of the former class version possibly causing phantom objects to exist in the updated application.

Table 2. Evaluation of run-time effects of performing specific code changes with Javeleon.

ID	Description of code change	Possible run-time phenomenon
1	Class removed	Phantom objects
2	Class added	Absent state
3	Modifier abstract added to class	Phantom objects
4	Super class of class changed	Absent state
5	Instance/static field added to class	Absent state
6	Modifier static removed from inner class	Absent state
7	Instance/static field type changed in class	Lost state
8	Static initialization impl. changed in class	Oblivious update
9	Constructor impl. changed in class	Oblivious update

Super class of class changed. When changing the super class of a class there may be fields in the chain of new super classes of which state is absent, because the constructor in the new super class has not been executed for already created objects, and some programmer-defined default field assignments have not been written properly to initialize those "new" fields.

Instance/static field added to class. Introducing a new field with a proper default value with respect to already initiated fields is not always possible, e.g., addition of SSN to Person class.

Modifier static removed from inner class. This code change will invalidate the language invariants of the new target code, since existing instances of the inner class, as of before the update, are created without the implicit strong association with the outer class. What happens is that after migrating existing instances across the version barrier, the new target code will have a generated synthetic field (along with a synthetic accessor method) for referencing the associated outer object. But, since the migrated objects did not contain this state information, trying to dereference the outer object will cause a null pointer exception.

Instance/Static field type changed in class. There are basically two kinds of field type changes: 1) compatible type changes not causing any problems and 2) incompatible type changes for which existing field values cannot be migrated to the new target code because they would not be assignable to the formal field type. Conceptually, any incompatible field type change should be regarded as a sequence of the two atomic code changes: *Field removed* from class and *Field added* to class.

Static initialization impl. changed in class, and Constructor impl. changed in class. Corresponds directly to that of the Oblivious Update.

Many developers do not think in terms of atomic code changes; rather they perform well-known refactoring, [10]. Therefore, we have conducted an investigation of how the usage of refactoring may cause the phenomena that we have been reporting on in this paper. We have carefully studied the run-time effects of all 72 refactoring in [10]. The resulting set of 22 refactoring patterns which may cause one of the discussed run-time phenomena is shown in Table 3.

Table 3. Refactoring with possible unwanted run-time effects.

Refactoring	Cause	Effect	Solution
Change Unidirectional Association to Bidirectional	Field added	Absent state	State restructuring
Change Value to Reference	Field added	Lost state	State restructuring
Collapse Hierarchy	Class removed	Phantom objects	Phantom Obejct cleanup
Duplicate Observed Data	Field added	Absent state	State restructuring
Extract Class	Field removed/added	Lost state	State relocation
Extract Hierarchy	Modifier abstract added to class	Phantom objects Oblivious update	Type widening
Extract Subclass	Class added	Oblivious update	Type widening
Inline Class	Class removed Field removed/added	Phantom objects Lost state	Phantom Obejct cleanup State relocation
Introduce Local Extension	Class added	Oblivious update	Type widening
Move Field	Field removed/added	Lost state	State relocation
Push Down Field	Field removed/added	Lost state	State relocation
Separate Domain from Presentation	Field added	Absent state	State restructuring
Remove Middle Man	Field added	Absent state	State restructuring
Replace Array with Object	Field type changed	Lost state	State restructuring
Replace Conditional with Polymorphism	Modifier abstract added to class	Phantom objects	Type widening
Replace Data Value with Object	Field type changed	Lost state	State restructuring
Replace Delegation with Inheritance	Super class of class changed	Lost state	State restructuring
Replace Inheritance with Delegation	Field added	Absent state	State restructuring
Replace Subclass with Fields	Field added Class removed	Absent state Phantom objects	State relocation Phantom Obejct cleanup
Replace Type Code with Class	Field type changed	Lost state	State restructuring
Replace Type Code with State/Strategy	Field type changed	Lost state	State restructuring
Replace Type Code with Subclass	Class added	Oblivious update	Type Widening
Tease Apart Inheritance	Field added Class removed	Absent state Phantom objects	State restructuring Phantom Object cleanup

The table lists the refactoring together with their atomic code changes and the particular phenomena that they may cause when performed as part of a dynamic update.

Javeleon assumes that state can be mapped logically in a one-to-one pairing to achieve State Migration across class versions. However, such a mapping, although very convenient with respect to the transparency of the mechanism, is not always sufficient to bring state across to a new class version as part of refactoring. Moving up a notch on the flexibility scale brings us to State Relocation, meaning the mechanism of moving state from one location in an application's design to a different location, without changing the type of that state. Clearly, this is what is needed when e.g. a Move Field refactoring is applied. State relocation requires a programmer's intervention, a task that can be performed in Javeleon by means of the described self-encapsulation scheme and/or STF. While state relocation provides an adequate amount of flexibility for performing several kinds of refactoring, there are situations in which state relocation is insufficient. Here programmers must bring State Restructuring into play. State restructuring is defined as the mechanism to correctly assign a new target state by composing it from arbitrary sources in the formerly running version. Clearly, state restructuring comes in handy when performing a number of refactoring operations, e.g., replacing simple values with objects hereby changing the type of the state.

In the context of performing run-time refactoring it is feasible to distinguish type restructuring implications in two fractions: 1) Type Widening and 2) Type Elimination. With the terminology given in [27] Type Widening allows an object to be temporarily widened, i.e., transformed into an instance of a subclass. In a refactoring such as "extract subclass" where the motivation is to provide one or more specific sub-types for an existing class, the run-time effect in a language transparent dynamic updating system is that existing objects of the class will be migrated to corresponding new instances of the new super-class because the class names would match. However, the intention was to allow creation of sub-types where appropriate. That is, existing objects, matching the requirements for creating an instance of the subclass in the new target code, should have been promoted to the new subclass dynamically. Thus, we say that the type of those objects should be widened. As can be seen from Table 3, any refactoring that requires Type Widening introduces a subclass to substitute some or all instances of an existing class. On the contrary, Type Elimination is inherently present in any refactoring which contains a *Class removed* atomic code change that invalidates the type of the existing objects of that class. The cure is to eliminate the effects of phantom objects as described in section 4.1.

An important observation from Table 3 is that there is a class of refactorings that cause run-time phenomena because they change objects to become composites, wherein the references to the new compositions cannot be deduced from the present application state. Likewise, other refactorings cause run-time phenomena because they introduce a delegation or forwarding field, which cannot be automatically initialized. All of these refactorings either add a field, a new class or both to create a link to the changed/new elements. The run-time phenomena caused by the said refactorings can be quite easily solved using a lookup service, as done in the NetBeans Platform, [4], to obtain the missing references. So using a proven type-safe and decoupled communication mechanism in the application design does not only provide a clean solution to a modular architecture, it also serves as a logical tool for permitting many automatic and seamless run-time refactorings.

6. DISCUSSION

Generalizations on our experience suggest that a dynamic updating system that is language transparent is not transparent to application design. From this, we argue that developers need to acquire an ability to think more dynamically if they want to achieve safe dynamic updating. As a rule of thumb they should, as a minimum, be aware that the design of their application's logic should be based on a dynamic interpretation of the application and its context, and not on static assumptions. The fact that a dynamic updating system is not transparent to the application design implies that the success of dynamic updates on a series of releases of an arbitrary application developed prior to introducing a dynamic updating system depends on the quality of the design. From our experience we have observed that the use of a modern component framework, such as the NetBeans Platform, assists developers in writing more context-aware and maintainable code. Indeed, the use of declarative registrations of listener objects would eliminate many oblivious update and transient consistency phenomena, because the declarations can be refreshed automatically when a dynamic update is applied.

By repeating updates at arbitrary times, we observed that the occurrence of run-time phenomena might depend on the timing of the dynamic updates. In this vein, we found that a language transparent dynamic updating system is vulnerable to time-sensitive changes that alter the perceived view of the present application state. In a trivial example from the Breakout Game, the only point in time we could correctly change the number of points gained per brick regardless of the underlying application design is before any bricks have been hit. Trying to utilize "safe update points" in this situation becomes rather complicated, because the "correct" time to update depends on the application state, not just the execution path of the combined versions.

We have shown examples where the perceived application behavior of the updated application differs from what a complete application restart would have shown. However, in many circumstances, especially with respect to transient inconsistencies, the deviation is acceptable because the application will return to an acknowledged application state shortly thereafter.

From our experience, the presence of a dynamic updating system will indeed influence the developers' perception of correct application behavior. Nonetheless, we have found this not to be a problem, as the immediate feedback given to developers when using Javeleon during the development phase causes them to think more about the variability intrinsic to the dynamic behavior of their applications, thereby finding logical ways of dealing correctly with the observed run-time phenomena.

Our experiments have shown that some design guidelines are required. But, does this imply that dynamic updating might as well be explicit in the language? We don't think so, since not only would this increase the complexity of learning the language, but more importantly it will most likely not provide us with a 100% guarantee that no phenomena will appear after a dynamic update. Based on the opinion of our limited number of test-users, we don't believe that the recommended design guidelines will scare developers from using a language-transparent dynamic updating system. Perhaps in the beginning they will be reluctant. But, we are convinced that a language-transparent dynamic updating style provides such a huge benefit during application development that developers will quickly adapt to these design guidelines without even paying special attention to them.

When it comes to live updating in a real production system, the guarantees provided by a dynamic updating system must be stronger. The present state-of-the-art systems are not quite ready to enter that stage yet, but we are not too far away.

6.1 Relation to Schema Evolution in Database Systems

The various phenomena in relation to existing instances are much the same as when the schema of an OODB is updated. The work done in [5] presents a solution for upgrading objects in a persistent data store. The solution differs from ours since it requires developers to write object transfer functions to upgrade objects to the new version. Moreover, to the best of our knowledge in the work done on schema evolution in databases, there is always an implicit assumption that the applications that use the changed schema contain corrective code to deal with any version mismatch, e.g., an additional uninitialized field. Such assumptions cannot simply be made within the area of application-level dynamic updating. Furthermore, in [5] or any other related work that we know of, the problem of phantom objects is not mentioned. The Phantom Object phenomenon cannot occur in any mechanism involving de-serialization since those objects would simply be omitted when the re-created object graph is built, because the declaring class would not be found.

7. CONCLUSION

In this paper, we have identified a number of run-time phenomena that are inherent to dynamic updating of applications written in statically-typed object-oriented programming languages. The causes and effects of these run-time phenomena on the correctness of the updated application have been investigated, in a series of case studies, using our own dynamic updating system - Javeleon. Our investigation has shown that the run-time behavior of an updated application depends on the application design. Depending on the specific design, dynamic updating may result in a different run-time behavior than the traditional halt, redeploy and restart scheme. We have discussed the associated problems of the run-time phenomena caused by dynamic updating and how application developers can remedy their effect by following best-practice design guidelines. The work presented in this paper provides a first step in gaining a better understanding of the implications of supporting dynamic updating in Java.

8. ACKNOWLEDGMENTS

Our thanks to Sun Microsystems/Oracle and the Danish Research Council for their support of this research.

9. REFERENCES

[1] Arnold, J. and Kaashoek, M. F. 2009. Ksplice: automatic rebootless kernel updates. In Proceedings of the 4th ACM European Conference on Computer Systems (Nuremberg, Germany, April 01 - 03, 2009). EuroSys '09. ACM, New York, NY, 187-198. DOI= http://doi.acm.org/10.1145/1519065.1519085

[2] Bierman, G., Parkinson, M., and Noble, J. 2008. UpgradeJ: Incremental Typechecking for Class Upgrades. In Proceedings of the 22nd European Conference on Object-Oriented Programming (Paphos, Cypress, July 07 - 11, 2008). J. Vitek, Ed. Lecture Notes In Computer Science, vol. 5142. Springer-Verlag, Berlin, Heidelberg, 235-259. DOI= http://dx.doi.org/10.1007/978-3-540-70592-5_11

[3] Bloch J. 2008. Effective Java. Addison-Wesley Java series, The Java series. Addison-Wesley, 2008. ISBN: 0321356683.

[4] Boudreau, T., Tulach, J., Wielenga, G. 2007. In Rich Client Programming: Plugging into the NetBeans(TM) Platform. Chapter 4 and 5. Prentice Hall PTR 2007, ISBN-13: 978-0132354806.

[5] Boyapati, C., Liskov, B., Shrira L., Moh, C.H. and Richman S. 2003. Lazy modular upgrades in persistent object stores. SIGPLAN Not. 38, 11 (October 2003), 403-417. DOI=http://doi.acm.org/10.1145/949343.949341.

[6] Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. 2005. Towards a taxonomy of software change: Research Articles. J. Softw. Maint. Evol. 17, 5 (Sep. 2005), 309-332. DOI= http://dx.doi.org/10.1002/smr.v17:5

[7] Chen, H., Yu, J., Chen, R., Zang, B., and Yew, P. 2007. POLUS: A POwerful Live Updating System. In Proceedings of the 29th international Conference on Software Engineering (May 20 - 26, 2007). IEEE Computer Society, Washington, DC, 271-281. DOI= http://dx.doi.org/10.1109/ICSE.2007.65

[8] Dmitriev M.: Safe Evolution of Large and Long-Lived Java Applications. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2001.

[9] Duggan, D. 2001. Type-based hot swapping of running modules (extended abstract). In Proceedings of the Sixth ACM SIGPLAN international Conference on Functional Programming (Florence, Italy, September 03 - 05, 2001). ICFP '01. ACM, New York, NY, 62-73. DOI= http://doi.acm.org/10.1145/507635.507645

[10] Fowler, M., Beck, K. 1999. Refactoring: improving the design of existing code, Addison-Wesley, 1999, ISBN: 0201485672.

[11] Gregersen, A. R. and Jørgensen, B. N. 2009. Dynamic update of Java applications - balancing change flexibility vs. programming transparency. J. Softw. Maint. Evol. 21, 2 (Mar. 2009), 81-112. DOI= http://dx.doi.org/10.1002/smr.v21:2

[12] Gregersen, A. R. 2010. Extending NetBeans™ with dynamic update of active modules. PhD thesis, Faculty of Engineering, University of Southern Denmark.

[13] Gustavsson, J. 2003. A Classification of Unanticipated Run-time Software Changes in Java. In Proceedings of the international Conference on Software Maintenance (September 22 - 26, 2003). ICSM. IEEE Computer Society, Washington, DC, 4.

[14] Hayden, C.M., Hardisty, E.A., Hicks, M. and Foster, S.F. 2009. A Testing Based Empirical Study of Dynamic Software Update Safety Restrictions. Department of Computer Science Technical Report CS-TR-4949.

[15] Hicks, M. and Nettles, S. 2005. Dynamic software updating. ACM Trans. Program. Lang. Syst. 27, 6 (Nov. 2005), 1049-1096. DOI= http://doi.acm.org/10.1145/1108970.1108971

[16] Hjálmtýsson, G. and Gray, R. 1998. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In Proceedings of the Annual Conference on USENIX Annual Technical Conference (New Orleans, Louisiana, June 15 - 19, 1998). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 6-6.

[17] Kabanov, J. 2010. JRebel Tool Demo. In proceedings of Bytecode 2010, 5th Workshop on Bytecode Semantics, Verification, Analysis and Transformation. (March 27, 2010). http://bytecode2010.inria.fr/pproc.pdf, 71-76.

[18] Lanza, M., Ducasse, S., Gall, H., and Pinzger, M. 2005. CodeCrawler: an information visualization tool for program comprehension. In Proceedings of the 27th international Conference on Software Engineering (St. Louis, MO, USA,

May 15 - 21, 2005). ICSE '05. ACM, New York, NY, 672-673. DOI= http://doi.acm.org/10.1145/1062455.1062602

[19] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. 2006. Practical dynamic software updating for C. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada, June 11 - 14, 2006). PLDI '06. ACM, New York, 72-83. DOI= http://doi.acm.org/10.1145/1133981.1133991

[20] Neamtiu, I. and Hicks, M. 2009. Safe and timely updates to multi-threaded programs. In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (Dublin, Ireland, June 15 - 21, 2009). PLDI '09. ACM, New York, NY, 13-24. DOI= http://doi.acm.org/10.1145/1542476.1542479

[21] Neamtiu, I., Hicks, M., Foster, J. S., and Pratikakis, P. 2008. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Francisco, California, USA, January 07 - 12, 2008). ACM, New York, 37-49. DOI= http://doi.acm.org/10.1145/1328438.1328447

[22] Orso A., Rao A., Harrold M.J. 2002. A Technique for Dynamic Updating of Java Software. In Proceedings of the international Conference on Software Maintenance (October 03 - 06, 2002). ICSM. IEEE Computer Society, Washington, DC, 649.

[23] Previtali, S. C. and Gross, T. R. 2006. Dynamic Updating of Software Systems Based on Aspects. In Proceedings of the 22nd IEEE international Conference on Software Maintenance (September 24 - 27, 2006). ICSM. IEEE Computer Society, Washington, DC, 83-92. DOI= http://dx.doi.org/10.1109/ICSM.2006.23

[24] Previtali, S. C., Schäuble, M., and Gross, T. R. 2009. Revision classes for explicit versioning. In Proceedings of the Workshop on AOP and Meta-Data For Software Evolution (Genova, Italy, July 07 - 07, 2009). M. Oriol, W. Cazzola, S. Chiba, and G. Saake, Eds. RAM-SE '09. ACM, New York, NY, 1-6. DOI= http://doi.acm.org/10.1145/1562860.1562861

[25] Redmond, B. and Cahill, V. 2002. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. In Proceedings of the 16th European Conference on Object-Oriented Programming (June 10 - 14, 2002). B. Magnusson, Ed. Lecture Notes In Computer Science, vol. 2374. Springer-Verlag, London, 205-230.

[26] Sato Y., Chiba S. 2005. Loosely-separated "Sister" Namespaces in Java. In proceedings of ECOOP'05. Lecture Notes in Computer Science, Vol. 3586. Springer-Verlag, (2005). 49-70. DOI= http://dx.doi.org/10.1007/11531142_3

[27] Serrano, M. 1999. Wide Classes. In Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99), Rachid Guerraoui (Ed.). Springer-Verlag, London, UK, 391-415.

[28] Stoyle, G., Hicks, M., Bierman, G., Sewell, P., and Neamtiu, I. 2007. Mutatis Mutandis: Safe and predictable dynamic software updating. ACM Trans. Program. Lang. Syst. 29, 4 (Aug. 2007), 22. DOI= http://doi.acm.org/10.1145/1255450.1255455

[29] Subramanian, S., Hicks, M., and McKinley, K. S. 2009. Dynamic software updates: a VM-centric approach. SIGPLAN Not. 44, 6 (May. 2009), 1-12. DOI= http://doi.acm.org/10.1145/1543135.1542478

[30] Vandewoude Y., Ebraert P., Berbers Y., D'Hondt T.: Influence of type systems on dynamic software evolution. Technical Report CW410, KULeuven, Belgium (2005).

[31] VisualVM: https://visualvm.dev.java.net

[32] Würthinger, T., Wimmer,C. and Stadler, L. 2010. Dynamic code evolution for Java. In Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10). ACM, New York, NY, USA, 10-19. DOI=10.1145/1852761.1852764 http://doi.acm.org/10.1145/1852761.1852764